This Course Is About... (1/2)

Data structures: how to store data.
Examples: arrays, linked lists.
This course goes way beyond them.
But why? Why other data structures?
Because you need this course to graduate. :)

To speed up some kinds of lookups and updates.

This Course Is About... (2/2)

We will learn:

- Several data structures.
- Algorithms that look up and update them.
- Why the algorithms are correct.
- How fast the algorithms are.
  (Therefore what the data structures are good for.)
  Sometimes also how much space.
- Making up our own data structures and algorithms.
  (Or adapting from what we learned.)

The course cannot possibly cover all data structures!
There are a lot more you will have to learn outside this course.

Course Web Page

http://www.cs.utoronto.ca/~trebla/CSCB63-2014-Summer/

And Blackboard.

Big-Oh

“Linear search takes $O(n)$ time.”
“Binary search takes $O(\log(n))$ time.”
“Bubble sort takes $O(n^2)$ time.”

$n^2 + 2 \cdot n + 1 \in O(n^2)$
$n^2 + 2 \cdot n + 1 \notin O(n)$

??? What are they saying?

We now turn to stuff that will be on the exam!
Let me scare you a bit first...

(Assume: for all natural $n$, $f(n) \geq 0$, $g(n) \geq 0$.)

Let me scare you a bit first...

(Assume: for all natural $n$, $f(n) \geq 0$, $g(n) \geq 0$.)

(In this course, 0 is a natural number.)

Definition: $f(n) \in O(g(n))$ iff

there exists real $c > 0$, natural $n_0$ such that
for all natural $n \geq n_0$,

$f(n) \leq c \cdot g(n)$

Help! What is it saying?!

Let me tell some revisionist history...
How much time does this take?

```java
public boolean containsDuplicate(int[] a, int n) {
    boolean e = false;
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            if (a[i] == a[j]) { e = true; }
        }
    }
    return e;
}
```

On one computer and one compiler:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read i, j, n</td>
<td>2</td>
</tr>
<tr>
<td>read a[i]</td>
<td>3</td>
</tr>
<tr>
<td>write i, j</td>
<td>3</td>
</tr>
<tr>
<td>arithmetic</td>
<td>1</td>
</tr>
<tr>
<td>two-way branch</td>
<td>1 if continue, 2 if exit</td>
</tr>
<tr>
<td>loop back</td>
<td>1</td>
</tr>
</tbody>
</table>

Total: $12 \cdot n^2 + 9 \cdot n + 8$

How much time does this take?

```java
public boolean containsDuplicate(int[] a, int n) {
    boolean e = false;
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            if (a[i] == a[j]) { e = true; }
        }
    }
    return e;
}
```

On another computer and another compiler:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read i, j, n</td>
<td>1</td>
</tr>
<tr>
<td>read a[i]</td>
<td>5</td>
</tr>
<tr>
<td>write i, j</td>
<td>1</td>
</tr>
<tr>
<td>arithmetic</td>
<td>1</td>
</tr>
<tr>
<td>two-way branch</td>
<td>0 if continue, 2 if exit</td>
</tr>
<tr>
<td>loop back</td>
<td>1</td>
</tr>
</tbody>
</table>

Total: $9.5 \cdot n^2 + 2.5 \cdot n + 4$

How to be both Coarse and Rigorous?

To a theoretician:

- the j-loop: $D \cdot (n - i - 1)$
- the i-loop: $\sum_{i=0}^{n-1} (C + \text{the j-loop})$
- the whole: $B + \text{the i-loop}$

Total: $\frac{D}{2} \cdot n^2 + (C + \frac{D}{2}) \cdot n + B$

How to be both Coarse and Rigorous?

- $12 \cdot n^2 + 9 \cdot n + 8$
- $9.5 \cdot n^2 + 2.5 \cdot n + 4$
- $\frac{D}{2} \cdot n^2 + (C + \frac{D}{2}) \cdot n + B$

How to unify them and ignore machine differences?

What is common about them? Quadratic polynomials.

Poster-boy for quadratic polynomials: $n^2$.

How to rigorously say: the above functions are “like” $n^2$?
How to be both Coarse and Rigorous?

12 \cdot n^2 + 9 \cdot n + 8
9.5 \cdot n^2 + 2.5 \cdot n + 4
\frac{D}{2} \cdot n^2 + (C + \frac{D}{2}) \cdot n + B

How to unify them and ignore machine differences?

What is common about them? Quadratic polynomials.
Poster-boy for quadratic polynomials: \(n^2\).

How to rigorously say: the above functions are “like” \(n^2\)?
And cubic polynomials are “like” \(n^3\)?

And \(4 \cdot n \cdot \log_2(n) + 2 \cdot n + 10\) is “like” \(n \cdot \log_2(n)\)?
Etc., etc.

How to be both Coarse and Rigorous!

Watch this mathemagic:

\[
12 \cdot n^2 + 9 \cdot n + 8 \leq 12 \cdot n^2 + 9 \cdot n^2 + 8
= 21 \cdot n^2 + 8
\]

For all natural \(n \geq 8\):

\[
21 \cdot n^2 + 8 \leq 21 \cdot n^2 + n
\leq 21 \cdot n^2 + n^2
= 22 \cdot n^2
\]

There exists natural \(n_0\) such that
for all natural \(n \geq n_0\), \(12 \cdot n^2 + 9 \cdot n + 8 \leq 22 \cdot n^2\)

How to be both Coarse and Rigorous!

Watch this mathemagic:

\[
12 \cdot n^2 + 9 \cdot n + 8 \leq 12 \cdot n^2 + 9 \cdot n^2 + 8
= 21 \cdot n^2 + 8
\]

For all natural \(n \geq 8\):

\[
21 \cdot n^2 + 8 \leq 21 \cdot n^2 + n
\leq 21 \cdot n^2 + n^2
= 22 \cdot n^2
\]

There exists natural \(n_0\) such that
for all natural \(n \geq n_0\), \(12 \cdot n^2 + 9 \cdot n + 8 \leq 22 \cdot n^2\)

How to be both Coarse and Rigorous!

Watch this mathemagic:

\[
12 \cdot n^2 + 9 \cdot n + 8 \leq 12 \cdot n^2 + 9 \cdot n^2 + 8
= 21 \cdot n^2 + 8
\]

For all natural \(n \geq 8\):

\[
21 \cdot n^2 + 8 \leq 21 \cdot n^2 + n
\leq 21 \cdot n^2 + n^2
= 22 \cdot n^2
\]

There exists natural \(n_0\) such that
for all natural \(n \geq n_0\), \(12 \cdot n^2 + 9 \cdot n + 8 \leq 22 \cdot n^2\)

There exists real \(c > 0\), natural \(n_0\) such that
for all natural \(n \geq n_0\), \(12 \cdot n^2 + 9 \cdot n + 8 \leq c \cdot n^2\)
The Rise of Big-Oh

There exists real $c > 0$, natural $n_0$ such that
for all natural $n \geq n_0$, $12 \cdot n^2 + 9 \cdot n + 8 \leq c \cdot n^2$

$12 \cdot n^2 + 9 \cdot n + 8 \in O(n^2)$

The Rise of Big-Oh

There exists real $c > 0$, natural $n_0$ such that
for all natural $n \geq n_0$, $12 \cdot n^2 + 9 \cdot n + 8 \leq c \cdot n^2$

$12 \cdot n^2 + 9 \cdot n + 8 \in O(n^2)$

$9.5 \cdot n^2 + 2.5 \cdot n + 4 \in O(n^2)$

$\frac{D}{2} \cdot n^2 + (C + \frac{D}{2}) \cdot n + B \in O(n^2)$

My algorithm takes time in $O(n^2)$
My algorithm takes $O(n^2)$ time
My algorithm takes time on the order of $n^2$

The Rise of Big-Oh

There exists real $c > 0$, natural $n_0$ such that
for all natural $n \geq n_0$, $12 \cdot n^2 + 9 \cdot n + 8 \leq c \cdot n^2$

$12 \cdot n^2 + 9 \cdot n + 8 \in O(n^2)$

$9.5 \cdot n^2 + 2.5 \cdot n + 4 \in O(n^2)$

$\frac{D}{2} \cdot n^2 + (C + \frac{D}{2}) \cdot n + B \in O(n^2)$

My algorithm takes time in $O(n^2)$
My algorithm takes $O(n^2)$ time
My algorithm takes time on the order of $n^2$

$4 \cdot n \cdot \log_2(n) + 2 \cdot n + 10 \in O(n \cdot \log_2(n))$

The Rise of Big-Oh

There exists real $c > 0$, natural $n_0$ such that
for all natural $n \geq n_0$, $12 \cdot n^2 + 9 \cdot n + 8 \leq c \cdot n^2$

$12 \cdot n^2 + 9 \cdot n + 8 \in O(n^2)$

$9.5 \cdot n^2 + 2.5 \cdot n + 4 \in O(n^2)$

$\frac{D}{2} \cdot n^2 + (C + \frac{D}{2}) \cdot n + B \in O(n^2)$

My algorithm takes time in $O(n^2)$
My algorithm takes $O(n^2)$ time
My algorithm takes time on the order of $n^2$

$4 \cdot n \cdot \log_2(n) + 2 \cdot n + 10 \in O(n \cdot \log_2(n))$

The definition is scary sophisticated because it has to drop some information and carefully preserve some other.

Using Limits to Prove Big-Oh

Theorem: If $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ exists and is finite, then $f(n) \in O(g(n))$. 

```latex
\begin{align*}
\text{Theorem: If } \lim_{n \to \infty} \frac{f(n)}{g(n)} \text{ exists and is finite, then } f(n) & \in O(g(n)). \\
\text{The definition is scary sophisticated because it has to drop some information and carefully preserve some other.}
\end{align*}
```
Using Limits to Prove Big-Oh

Theorem: If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \) exists and is finite, then \( f(n) \in O(g(n)) \).

Prove: \( n \cdot (n + 1)/2 \in O(n^2) \)

\[
\lim_{n \to \infty} \frac{n \cdot (n + 1)/2}{n^2} = \frac{1}{2}
\]

Therefore \( n \cdot (n + 1)/2 \in O(n^2) \)

Prove: \( n \in O(n^2) \)

Prove: \( \ln(n) \in O(n) \)

Using Limits to Prove Big-Oh

Theorem: If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \) exists and is finite, then \( f(n) \in O(g(n)) \).

Prove: \( n \cdot (n + 1)/2 \in O(n^2) \)

\[
\lim_{n \to \infty} \frac{n \cdot (n + 1)/2}{n^2} = \frac{1}{2}
\]

Therefore \( n \cdot (n + 1)/2 \in O(n^2) \)

Prove: \( n \in O(n^2) \)

\[
\lim_{n \to \infty} \frac{n}{n^2} = \lim_{n \to \infty} \frac{1}{n} = 0
\]

Therefore \( n \in O(n^2) \)

Prove: \( \ln(n) \in O(n) \)

Using Limits to Disprove Big-Oh

Theorem: If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \), then \( f(n) \notin O(g(n)) \).

Disprove: \( n^2 \in O(n) \)

\[
\lim_{n \to \infty} \frac{n^2}{n} = \lim_{n \to \infty} n = \infty
\]

Therefore \( n^2 \notin O(n) \)

Disprove: \( n \in O(\ln(n)) \)

Using Limits to Disprove Big-Oh

Theorem: If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \), then \( f(n) \notin O(g(n)) \).

Disprove: \( n^2 \in O(n) \)

\[
\lim_{n \to \infty} \frac{n^2}{n} = \lim_{n \to \infty} n = \infty
\]

Therefore \( n^2 \notin O(n) \)

Disprove: \( n \in O(\ln(n)) \)

\[
\lim_{n \to \infty} \frac{n}{\ln(n)} = \lim_{n \to \infty} \frac{1}{\ln(n)} = \infty
\]

Therefore \( n \notin O(\ln(n)) \)
When Limits Don’t Help

Theorem: If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \) exists and is finite, then . . .

Theorem: If \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \), then . . .

Which case has not been covered?

Examples When Limits Don’t Help

Define: \( \text{drunk}(n) = \) if \( n \) is even then 1 else \( n \)

(Plot it to see what it looks like!)

Example:

\[ \lim_{n \to \infty} \frac{\text{drunk}(n)}{n} \]

does not exist and is not \( \infty \), and still \( \text{drunk}(n) \in O(n) \).

Example:

\[ \lim_{n \to \infty} \frac{\text{drunk}(n)}{1} \]

does not exist and is not \( \infty \), and this time \( \text{drunk}(n) \notin O(1) \)

About \( n \in O(n^2) \)

My friend says: “my algorithm takes \( O(n^2) \) time.”

- It may take \( 9 \cdot n^2 + 4 \cdot n + 13 \) time.

About \( n \in O(n^2) \)

My friend says: “my algorithm takes \( O(n^2) \) time.”

- It may take \( 9 \cdot n^2 + 4 \cdot n + 13 \) time.
- It may take \( 3 \cdot n + 2 \) time.
  \( 3 \cdot n + 2 \in O(n^2) \) is still true.
About $n \in O(n^2)$

My friend says: “my algorithm takes $O(n^2)$ time.”

- It may take $9 \cdot n^2 + 4 \cdot n + 13$ time.
- It may take $3 \cdot n + 2$ time.
  $3 \cdot n + 2 \in O(n^2)$ is still true.
- It may take 45 time.
  $45 \in O(n^2)$ is still true.

About $n \in O(n^2)$

My friend says: “my algorithm takes $O(n^2)$ time.”

- It may take $9 \cdot n^2 + 4 \cdot n + 13$ time.
- It may take $3 \cdot n + 2$ time.
  $3 \cdot n + 2 \in O(n^2)$ is still true.
- It may take 45 time.
  $45 \in O(n^2)$ is still true.

$O(n^2)$ includes quadratic functions as well as “lesser” functions.

We need another definition to exclude “lesser” functions.

Draw inspiration from these:

- $n^2 \in O(9 \cdot n^2 + 4 \cdot n + 13)$
- $n^2 \in O(3 \cdot n + 2)$
- $n^2 \in O(45)$

Big-Theta, Θ

Definition: $f(n) \in \Theta(g(n))$ iff

$f(n) \in O(g(n))$ and $g(n) \in O(f(n))$

Example: $9 \cdot n^2 + 4 \cdot n + 13 \in \Theta(n^2)$

Example: $n \notin \Theta(n^2)$
  since $n^2 \notin O(n)$

Example: $\ln(n) \notin \Theta(n)$
  since $n \notin O(\ln(n))$

Big-Theta, Θ

Definition: $f(n) \in \Theta(g(n))$ iff

$f(n) \in O(g(n))$ and $g(n) \in O(f(n))$

Example: $9 \cdot n^2 + 4 \cdot n + 13 \in \Theta(n^2)$

Example: $n \notin \Theta(n^2)$
  since $n^2 \notin O(n)$

Example: $\ln(n) \notin \Theta(n)$
  since $n \notin O(\ln(n))$

My opinion: You should say Θ as much as you can.

Popular opinion: Just say $O$.

A good use of $O$: Assignment or contract says:

"Write a $O(n^2)$-time algorithm."

It allows you to do better.

Big-Oh, Big-Theta May Miss Something

$10^{100} \cdot n \in \Theta(n)$

$n + 10^{100} \in \Theta(n)$

Can’t say these are practical algorithm times.
But big-Oh, big-Theta can’t detect them.
This is a price for ignoring machine differences.

These pathological cases are rare.
Big-Oh, big-Theta are usually informative.

Big-Oh, Big-Theta Gain Simplification

```java
e = false;
for (i = 0; i < n; i++) {
    for (j = i+1; j < n; j++) {
        if (a[i] == a[j]) { e = true; }
    }
}
```

"12 \cdot n^2 + 9 \cdot n + 8 \text{ ns}"

Took me 15 minutes to figure out.

"The inner loop takes $O(n - i - 1)$ time, which is $O(n)$.
The outer loop takes $n$ iterations.
Altogether $O(n^2)$ time."

This only takes 30 seconds to figure out.

It is one of the reasons we use big-Oh, big-Theta.
Worse Case, Best Case

```java
  e = false;
  for (i = 0; i < n; i++) {
    for (j = i+1; j < n; j++) {
      if (a[i] == a[j]) { e = true; break; }
    }
  }
```

Depending on what's in a:
Anywhere from $\Theta(1)$ time to $\Theta(n^2)$ time.

Best case is $\Theta(1)$ time.
Worse case is $\Theta(n^2)$ time.

We look at worst cases in this course mostly.

Myth Buster

Myth: Big-Oh means worst case time.

Truth: Big-Oh, big-Theta compare functions, do not say what the functions are for.

$9 \cdot n^2 + 4 \cdot n + 13$ may be best case time, or worst case time, or best case space, or worst case space, or just a polynomial from nowhere.

$9 \cdot n^2 + 4 \cdot n + 13 \in O(n^2)$ is true regardless.

"Best case time is in $O(n^2)$" is allowed.

Big-Oh, big-Theta are good for any function from natural to non-negative real.

Storing a Set

I want to store a set of keys.

I'm given: keys can be compared (<, =, >).

I want these operations to be fast:

- `s.find(x)`: return whether x is in s
- `s.insert(x)`: add x to s
- `s.remove(x)`: delete x from s

Binary Search Tree: Introduction

```java
  public class Node<K extends Comparable<K>> {
    public K key;
    public Node <K> left , right;
    public Node(K x) { key = x; left = null; right = null; }
  }
```

Why would you store data this way?

Storing a Set: Near Miss

(Reminder: Keys can be compared (<, =, >).)

If I wanted this only:

- `s.find(x)`: return whether x is in s

then you already know how to do it.
Storing a Set: Near Miss

(Reminder: Keys can be compared (<, =, >).)
If I wanted this only:
> s.find(x): return whether x is in s
then you already know how to do it.
Store in an array in increasing order:

```
8 13 20 42 47 85 91
```
Use binary search for find. $O(\log_2(n))$ time.

But insert and remove are icky.

Storing a Set: Binary Search Tree

A binary search tree:
> is a binary tree
> at each node v:
  v.key is greater than all keys in the left sub-tree
  v.key is smaller than all keys in the right sub-tree

The textbook ignores the empty tree.
I include the empty tree.

```
8 13 20 42 47 85 91
```

```
   42
  /   \
13    85
 /  \\  /
8 20 47 91
```

Good news for s.find(x):
left child, right child are exactly what you would try next in binary search!

Binary Search Tree: find

```java
private Node<K> root;
...
public boolean find(K x) {
    Node<K> v = root;
    while (v != null) {
        int r = v.key.compareTo(x);
        if (r == 0) {
            return true;
        } else if (r > 0) {
            v = v.left;
        } else {
            v = v.right;
        }
    }
    return false;
}
```

Binary Search Tree: insert

```java
public void insert(K x) {
    if (root == null) {
        root = new Node<K>(x);
    } else {
        Node<K> v = root;
        for (;;) {
            if (v.key.compareTo(x) == 0) {
                return;
            } else if (v.key.compareTo(x) > 0) {
                if (v.left == null) {
                    v.left = new Node<K>(x);
                    return;
                } else {
                    v = v.left;
                }
            } else {
                // similar, except it's v.right
            }
        }
    }
}
```
Start with the empty tree. Insert $1, 2, \ldots, n$ in that order. What do you get?

The tree is leaned on one side. Find, insert, remove are not $O(\log_2(n))$ anymore. What to do? What to do?

Cliff hanger! Come next time for a solution!